Project 2: Efficient Dense Matrix Multiplication

This project weights 12.5% for your final grade (4 Projects for 50%)

Release Date:

October 10th，2023 (Beijing Time, UTC+08:00)

Deadline:

11:59 P.M., October 23rd, 2023 (Beijing Time, UTC+08:00)

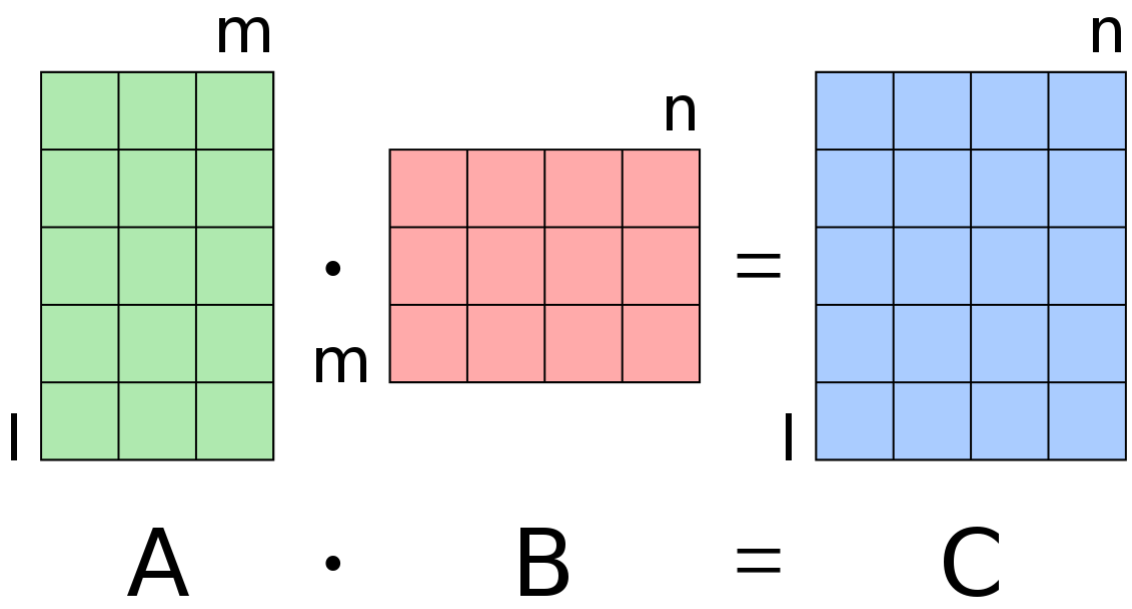
TA In Charge of This Assignment

Mr. Yang Yufan for Parallel Program Implementation (yufanyang1@link.cuhk.edu.cn)

Mr. Liu Yuxuan for Profiling (yuxuanliu1@link.cuhk.edu.cn)

Prologue

For the second programming project, you are tasked with **implementing an efficient dense matrix multiplication** algorithm based on the techniques covered in this course. Matrix multiplication plays an increasingly important role in today's AI landscape, serving as a fundamental component of deep neural networks (DNNs), particularly with the development of large language models (LLMs). Matrix multiplication optimization has been a classic research topic for algorithms for decades, resulting in many classic algorithms tailored to different matrix formats. This project, however, is relatively straightforward, as we focus solely on the most general scenario: **dense matrix multiplication**. The goal of this project is to help you **apply the optimization techniques taught in the course to improve dense matrix multiplication performance**.



At the outset, you will receive a poorly implemented dense matrix multiplication function, and your task is to **optimize it systematically**, considering factors such as **memory locality, SIMD (Single Instruction, Multiple Data), thread-level parallelism, and process-level parallelism** step by step. In your report, you should **document the performance improvements achieved after applying each technique**. Ultimately, you are expected to **submit a program that incorporates all of the aforementioned optimization techniques**, and we will evaluate whether the performance of your implementation meets our expectations.

Task1: Memory Locality

We have provided you with a basic sequential implementation of dense matrix multiplication as follows.

```
size_t M = matrix1.getRows(), K = matrix1.getCols(), N = matrix2.getCols();

Matrix result(M, N);

for (size_t i = 0; i < M; ++i) {
    for (size_t j = 0; j < N; ++j) {
        for (size_t k = 0; k < K; ++k) {
            result[i][j] += matrix1[i][k] * matrix2[k][j];
        }
    }
}
```

This implementation suffers from poor performance due to its suboptimal memory locality. In Task 1, you are required to:

1. In your report, analyze why our provided implementation of dense matrix multiplication exhibits such poor performance.
2. Complete the `Matrix matrix_multiply_locality(const Matrix& matrix1, const Matrix& matrix2)` function in `src/locality.cpp`. Your goal is to optimize our implementation by enhancing memory locality.
    i. Note: You **cannot** apply any parallel techniques at this stage.
    ii. Hint: Here are some methods you may try to increase memory locality and avoid cache misses:
        a. Change the order of the triple loop
        b. Apply tiled matrix multiplication
3. In your report, demonstrate the performance improvement achieved after implementing your changes.

Task2: Data-Level Parallelism

After completing Task 1, you should already have a relatively efficient sequential implementation for dense matrix multiplication. However, this level of efficiency is not sufficient. In fact, Single Instruction, Multiple Data (SIMD) techniques have been widely employed in many high-performance matrix multiplication libraries. Therefore, in Task 2, you are tasked with:

1. Completing the `Matrix matrix_multiply_simd(const Matrix& matrix1, const Matrix& matrix2)` function in `src/simd.cpp`. Your goal is to further enhance your implementation of dense matrix multiplication by applying SIMD techniques.
    i. **Note: you should build upon the work from Task 1**.
2. In your report, showcasing the performance improvement achieved after implementing these changes

Task3: Thread-Level Parallelism

Now is the time to introduce thread-level parallelism to your implementation to further enhance the efficiency of dense matrix multiplication. We recommend utilizing OpenMP for its user-friendly approach. Therefore, in Task 3, you have the following objectives:

1. Complete the `Matrix matrix_multiply_openmp(const Matrix& matrix1, const Matrix& matrix2)` function in `src/openmp.cpp`. Your goal is to expand the application of thread-level parallelism to

your implementation using OpenMP.
    i. **Note: You should build upon the work from Task 2.**
2. In your experiments, vary the thread num from 1, 2, 4, 8, 16 to 32, and observe the performance improvements.
3. In your report, showcase the performance improvements achieved after implementing these changes.

Task4: Process-Level Parallelism

Finally, you are tasked with introducing process-level parallelism to your dense matrix multiplication implementation to further enhance efficiency, utilizing MPI. Therefore, in Task 4, you should:

1. Complete the `Matrix matrix_multiply_mpi(const Matrix& matrix1, const Matrix& matrix2)` function in `mpi.cpp` to extend the application of process-level parallelism to your implementation using MPI. Additionally, you need to edit the `main` function to incorporate other MPI-specific logic, such as process communication.
    i. **Note: You should build upon the work from Task 3.**
2. In your experiments, keep the total thread num fixed at 32 (where process num * thread num per process = 32), but adjust the process num and thread num per process accordingly (1 x 32, 2 x 16, 4 x 8, 8 x 4, 16 x 2, 32 x 1). Observe the performance changes.
3. In your report, demonstrate the performance improvements (if any) achieved after implementing these changes.

Extra Credits: GPU Matrix Multiplication

Nowadays, GPUs are widely applied in AI areas to accelerate matrix multiplication. Therefore, we have chosen GPU Matrix Multiplication as the bonus task for Project 2.

For the bonus task, you can choose either CUDA or OpenACC to implement GPU Matrix Multiplication by completing `src/gpu/cuda.cu` or `src/gpu/openacc.cpp`. We haven't provided a code skeleton for this task because there are many optimization techniques available for GPU Matrix Multiplication, and you have the freedom to choose your approach. We expect to see a significant performance improvement compared to the CPU version.

As part of the bonus task, you should submit detailed instructions in your report on how to compile and execute your program. Additionally, you are required to showcase and analyze the performance improvements compared to the best CPU implementation.

Requirements & Grading Policy

- **Six parallel programming implementations for PartB (60%)**
  - Task1: Memory Locality (15%)
  - Task2: Data-Level Parallelism (15%)
  - Task3: Thread-Level Parallelism (15%)
  - Task4: Process-Level Parallelism (15%)

  Your programs should be able to compile & execute to get the computation result. Besides, to judge the correctness of your program, we will prepare 5 matrix testcases, you can get full mark if your programs pass all of them. Each failure of testcase will cause a 3-point deduction on that Task.
- **Performance of Your Program (20%)**

  Try your best to do optimization on your parallel programs for higher speedup. If your programs shows similar performance to the baseline performance, then you can get full mark for this part. Points will be deduted if your parallel programs perform poor while no justification can be found in the report. The 20% point will be divided into 4 x 5% for each of the four tasks.
- **One Report in PDF (20%, No Page Limit)**
  - **Regular Report As Project-1 (10%)**

    The report does not have to be very long and beautiful to help you get good grade, but you need to include what you have done and what you have learned in this project. The following components should be included in the report:
    - How to compile and execute your program to get the expected output on the cluster.
    - Briefly explain how does each parallel programming model do computation in parallel?
    - What kinds of optimizations have you tried to speed up your parallel program, and how does them work?
    - Show the experiment results you get, and do some numerical analysis, such as calculating the speedup and efficiency, demonstrated with tables and figures.
    - What have you found from the experiment results?
  - **Profiling Results & Analysis with `perf` (10%)**

    Please follow the Instruction on Profiling with perf and nsys to profile all of your parallel programs for the four tasks with `perf`, and do some analysis on the profiling results before & after the implementation or optimization. For example, for Task 1, you are asked to optimize the memory access pattern, decreasing cache misses and page faults for better efficiency. You can use the profiling results from `perf` to do quantitative analysis that how many cache misses or page faults can be reduced with your optimization. Always keep your mind open, and try different profiling metrics in `perf` and see if you can find any interesting thing during experiment.

Note: The raw profiling results may be very long. Please extract some of the useful items to show in your report, and remember to carry all the raw profiling results for your programs when you submit your project on BB.

- **Extra Credits (10%)**
  - CUDA Implementation (10%) or OpenACC Implementation (10%)
  - If you can pass all the testcases we provide, you can get the points. Optimized performance is not required. Choose one from CUDA and OpenACC to implement is enough to get full mark for extra credits.
  - Any interesting discoveries or discussions regarding the experiment results.

The Extra Credit Policy

According to the professor, the extra credits in project 1 cannot be added to other projects to make them full mark. The credits are the honor you received from the professor and the teaching stuff, and the professor may help raise you to a higher grade level if you are at the boundary of two grade levels and he think you deserve a better grade with your extra credits. For example, if you are the top students with B+ grade, and get enough extra credits, the professor may raise you to A- grade.

Grading Policy for Late Submission

1. late submission for less than 10 minutes after then DDL is tolerated for possible issues during submission.
2. 10 Points deduction for each day after the DDL (11 minutes late will be considered as one day, so be careful)
3. Zero point if you submitted your project late for more than two days
   If you have some special reasaons for late submission, please send email to the professor and c.c to TA Liu Yuxuan.

File Structure to Submit on BlackBoard

```
118010200.zip
|-
|--- 118010200.pdf  # Report
|-
|--- src/           # Where your source codes lie in
|--- matrices/      # Eight matrix testcases
|--- CMakeLists.txt # Root CMakeLists.txt
|-
|--- profiling/     # Where your perf profiling raw results lie in
```

How to Execute the Program

Data Set

We have provided 4 groups of matrices for your testing under `/path/to/project2/matrices`:

Group	Size	MatrixA	MatrixB
1	4*4	matrix1.txt	matrix2.txt
2	128*128	matrix3.txt	matrix4.txt
3	1024*1024	matrix5.txt	matrix6.txt
4	2048*2048	matrix7.txt	matrix8.txt

Only matrices within the same group can be multiplied together. We recommend using Group 1 matrices to verify the correctness of your matrix multiplication implementation. For performance testing, it's better to use matrices from Groups 3 and 4 to clearly observe the performance improvements.

Compilation

```
cd /path/to/project2
mkdir build && cd build
# Change to -DCMAKE_BUILD_TYPE=Debug for debug build error message logging
# Here, use cmake on the cluster and cmake3 in your docker container
cmake ..
make -j4
```

Compilation with `cmake` may fail in docker container, if so, please compile with `gcc`, `mpic++`, `nvcc` and `pgc++` in the terminal with the correct optimization options.

Local Execution

```
cd /path/to/project2/build
# Naive
./src/naive /path/to/matrixA /path/to/matrixB /path/to/multiply_result
# Memory Locality
./src/locality /path/to/matrixA /path/to/matrixB /path/to/multiply_result
# SIMD
./src/simd /path/to/matrixA /path/to/matrixB /path/to/multiply_result
# OpenMP
./src/openmp $thread_num /path/to/matrixA /path/to/matrixB /path/to/multiply_result
# MPI
mpirun -np $process_num ./src/mpi $thread_num_per_process /path/to/matrixA /path/to/matrixB /pa
```

Job Submission

Important: Change the directory of output file in `sbatch.sh` first, and you can also change the matrix files for different testing.

```
# Use sbatch
cd /path/to/project2
sbatch ./src/sbatch.sh
```

Performance Baseline

Experiment Setup

- On the cluster, allocated with 32 cores
- Matrices 1024 * 1024 with `matrix5.txt` and `matrix6.txt`
- Matrices 2048 * 2048 with `matrix7.txt` and `matrix8.txt`
- Use `src/sbatch.sh`

Methods	Matrices 1024*1024	Matrices 2048*2048
Naive	8165 ms	89250 ms
Memory Locality	776 ms	6509 ms

Methods	Matrices 1024*1024	Matrices 2048*2048
SIMD + Memory Locality	273 ms	2720 ms
OpenMP + SIMD + Memory Locality (32 threads)	41ms	214 ms
MPI + OpenMP + SIMD + Memory Locality (total 32 threads)	33 ms	184 ms